

- PROJET S6 -
Le jeu des Amazones

-

Mathias ROSA, Mohamed REKIK,
Mathias APARICIO, Hugo BASTIEN

12 mai 2023



Table des matières

1	Introduction	3
2	Rendu du projet	4
3	Interaction client - serveur	5
4	Implémentation du jeu	6
4.1	Graphe de dépendances	6
4.2	Interface commune au serveur et au joueur	6
5	Gestion du jeu	7
5.1	Initialisation	7
5.2	Boucle de jeu	8
6	Stratégie de jeu choisie	9
7	Contrainte de projets et travail collaboratif	10
7.1	Git et Makefile	10
7.2	Répartition du travail	11
8	Conclusion	11

1 Introduction

Le projet se situe dans l'UE projets du semestre 6 et permet notamment d'appliquer les enseignements reçus en cours de programmation impérative 2 et de programmation fonctionnelle. La durée était de 7 semaines répartis en séances de 4h20 les vendredis après midi.

Le sujet du projet est le jeu des amazones. C'est un jeu de plateau type échiquier, où deux joueurs s'affrontent en déplaçant des reines. Lors qu'une reine se déplace elle doit tirer une flèche sur une case libre du plateau, se trouvant à sa portée. Les cases accessibles par une amazone et par une flèche à partir de cette dernière sont les cases se trouvant dans les 8 directions cardinales, comme la dame aux échecs.

Pendant les séances, **Romain LION** nous a apporté son aide et nous le remercions chaleureusement.



FIGURE 1 – Le jeu d'amazones

2 Rendu du projet

Le projet peut simuler des parties à deux joueurs jouant au jeu des amazones sur une grille d'une largeur minimale de 5 cases. L'affichage du plateau se fait sur le terminal, comme le montre les figures 2, 3, 4.

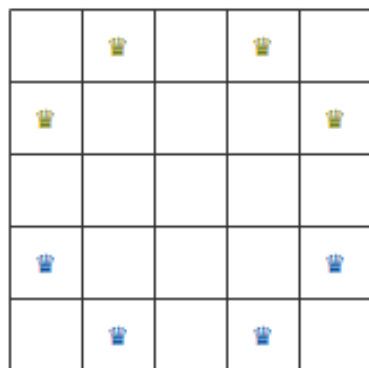


FIGURE 2 – Grille initiale pour une largeur de 5 cases

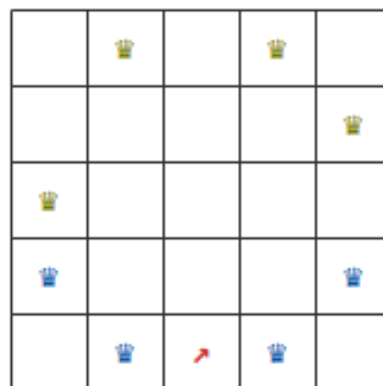


FIGURE 3 – Grille après le coup du premier joueur

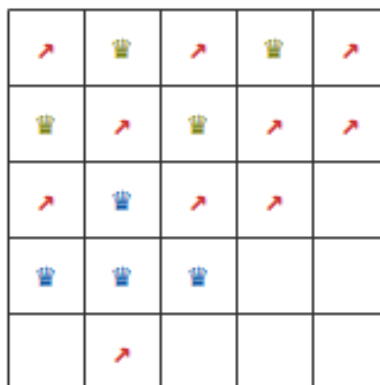


FIGURE 4 – Grille en fin de jeu, le joueur bleu a triomphé

Le programme accepte plusieurs paramètres dans la ligne de commande comme la taille du plateau de jeu qui est variable, ou le type de la grille. L'option `-m` permet d'initialiser à l'exécution la largeur du plateau en renseignant le nombre de cases en largeur, de même l'option `-t` permet de choisir la forme de la grille : carrée ou donut. La grille donut est quasiment identique à la grille carrée au détail qu'il y a un groupe de cases inaccessibles au centre du carré. Si

l'option de la grille donut est activé, le nombre de cases en largeur doit être un multiple de 3. Les figures 5 et 6 le montre.

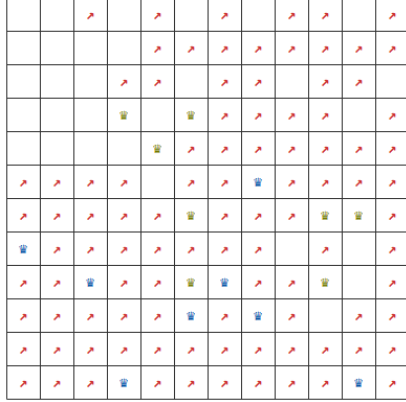


FIGURE 5 – Grille finale carré pour une largeur de 12 cases

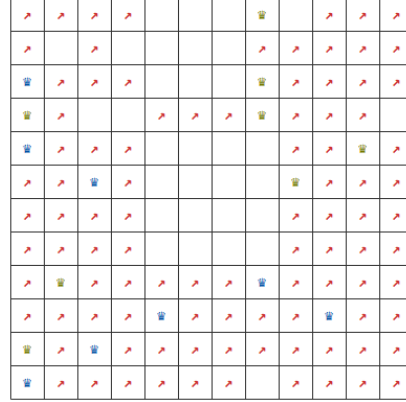


FIGURE 6 – Grille finale en forme de donut pour 12 cases de largeur

3 Interaction client - serveur

Le serveur simule une partie entre deux joueurs, où chacun a la possibilité de choisir son coup. Pour cela, chaque joueur dispose de sa propre grille qu'il peut modifier selon sa stratégie. Les fichiers des différents joueurs sont compilés en une bibliothèque partagée.

Les bibliothèques partagées permettent aux joueurs et au serveur d'exécuter leurs coups de manière indépendante. Cela nécessite que chaque joueur possède des fonctions spécifiques qui seront appelées par le serveur pour lancer le jeu et faire jouer chaque joueur à tour de rôle.

Le chargement des fonctions présentes dans les bibliothèques dynamiques se fait grâce à la fonction *dlopen*. Ensuite, en utilisant *dlsym*, on attribue un nom à une fonction spécifique dans les bibliothèques afin de la localiser dans la mémoire. Tout cela permet au serveur d'utiliser ces fonctions pour faire fonctionner le jeu.

À la fin du jeu, le serveur appelle *dlclose* pour libérer les ressources associées à la bibliothèque.

4 Implémentation du jeu

4.1 Graphe de dépendances

Pour utiliser dans un fichier des constantes et des fonctions déclarées dans d'autres fichiers on a besoin d'inclure les "headers" de ces autres fichiers. Ces inclusions ne doivent pas être redondantes. Ces dépendances sont observables sur la Figure 7.

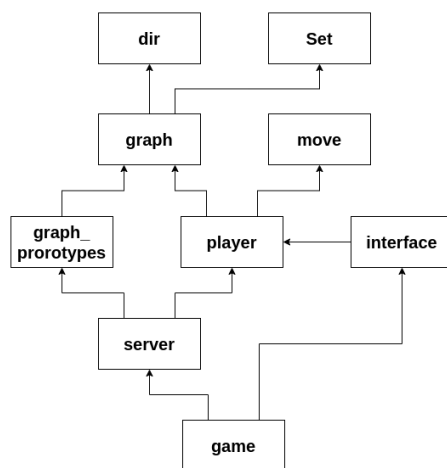


FIGURE 7 – Le graphe des dépendances.
Une flèche de A vers B signifie A inclut B.

4.2 Interface commune au serveur et au joueur

Il a été décidé pour éviter la duplication de code d'écrire un fichier implémentant des primitives utilisé aussi bien par le serveur que par le joueur. Par exemple la fonction `get_neighboors` permet de récupérer les coups jouables par une dame, cela permet donc au joueur de choisir un coup et cela permet au serveur de vérifier si un coup est légal.

Afin de manipuler le plateau du jeu, on s'est servi de la matrice d'adjacence liée au plateau donnant les voisins "géométriques" de chaque case, et d'un double tableau `queens` stockant les positions de chaque dame.

Intéressons-nous d'abord à comment est implémentée la matrice d'adjacence, puis comment était gérée la recherche de coups avec les structures sus mentionnées.

La matrice d'adjacence est représentée par une matrice où le coefficient $a_{i,j}$ de la matrice indique la direction pour aller de la case i à j si elles sont adjacentes, et 0 sinon. Ainsi les voisins de la case i sont simplement les valeurs non nulles de la ligne i dans cette matrice. Le nombre de cases étant grand devant le nombre de voisins d'une case, le choix des matrices creuses se justifie.

En effet l'implémentation des matrices creuses avec la librairie de calcul scientifique GNU GSL est pratique. Dans un format quelconque, accéder aux voisins d'une case demande d'itérer sur toute une ligne soit le nombre de cases. Alors que dans le format où le travail nous invitait à travailler le nombre d'itérations est le nombre de voisins directement.

Le principe est le suivant on travaille avec le type CSR, tous les éléments non nuls de la matrice sont stocké dans un tableau data et on a accès aux pointeurs de début de chaque ligne dans data.

Cependant, pour récupérer les mouvements, il faut savoir où sont les dames et si les cases disponibles géométriquement ne sont pas bloqués physiquement par une autre dame ou une flèche. Les positions des dames sont donc simplement stockée dans queens, et les flèches sont stockées dans un tableau arrow.

En conclusion chaque joueur utilise l'interface pour obtenir les coups jouables et les envoyer tout en mettant à jour son propre plateau pour avoir une vue sur la partie. Le serveur peut lui jouer les coups les vérifier et mettre à jour aussi.

5 Gestion du jeu

5.1 Initialisation

L'initialisation commence par la gestion des options qui permet de récupérer les paramètres de la partie.

Ensuite, les joueurs sont chargés dans un ordre aléatoire, le premier joueur chargé étant le premier à jouer par la suite. Plusieurs graphes sont initialisés : une fonction renvoie un pointeur vers une structure *graph_t* contenant la matrice d'adjacence du graphe. La matrice d'adjacence étant une matrice creuse, elle se trouve sous le format CSR, ce qui réduit considérablement l'espace de stockage nécessaire. Le format CSR a la particularité de ne stocker que les coefficients non nuls et d'indiquer à quelle ligne et à quelle colonne ils appartiennent. Un point délicat de l'initialisation du jeu est le placement des reines. En effet, les reines ont des positions initiales qui varient en fonction de leur nombre et donc de la largeur du plateau. La fonction *queens_init* permet d'initialiser un tableau de type *unsigned int *** avec les positions initiales des reines. Le tableau de queens est alloué dans le TAS et chaque sous-tableau l'est également. Ainsi chaque joueur reçoit un tableau de reines identique mais à des adresses différentes pour éviter de potentiels problèmes causés par les joueurs qui ont le droit de modifier ce tableau.

Le placement se fait à l'aide d'une boucle calculant l'indice de la case à laquelle la reine doit commencer la partie. Le calcul parait compliqué mais il est très simple. Pour chaque reine, on définit l'origine à partir de laquelle on commence à compter puis le sens de parcours : Si l'identifiant du joueur est 1, on commence par la fin de la grille, et le sens de parcours est positif si l'origine est 0, et négative sinon. Les formules sont détaillées sur l'extrait de la fonction d'initialisation suivant.

```

1 // INITIALISATION DU TABLEAU **QUEENS
2 unsigned int width = sqrt(num_vertices);
3 for (unsigned int player_id = 0; player_id < NUM_PLAYERS;
4     player_id++)
5 {
6     unsigned int origine = ((num_vertices - 1) * player_id);
7
8     for (unsigned int queen_id = 0; queen_id < num_queens / 4;
9         queen_id++)
10     {
11         int sens_parcours = (2 * player_id - 1) * (-1);
12
13         // ligne horizontale pour chaque joueur
14         queens[player_id][2 * queen_id] = origine +
15         sens_parcours * ((width / (num_queens / 2)) * queen_id + 1);
16
17         queens[player_id][2 * queen_id + 1] = origine +
18         sens_parcours * ((width - 1) - ((width / (num_queens / 2)) *
19         queen_id + 1));
20
21         // ligne verticales
22         queens[player_id][num_queens / 2 + 2 * queen_id] =
23         origine + sens_parcours * (width * ((width / (num_queens / 2))
24         * queen_id + 1));
25
26         queens[player_id][num_queens / 2 + 2 * queen_id + 1] =
27         origine + sens_parcours * (width * ((width / (num_queens / 2))
28         * queen_id + 1) + width - 1);
29     }
30 }

```

À la première itération, *player_id* est égal à 0, donc l'origine est la case 0, le sens de parcours est positif et le décalage est d'une case pour la première reine, la (largeur - 1) moins une case pour la seconde. De même pour les deux autres queens. Ensuite, *player_id* vaut 1 donc l'origine vaut *num_vertices - 1* qui correspond à la case 24 et le sens de parcours est négatif donc de la même façon que précédemment la première reine est décalée d'une case vers la droite donc en case 23, comme on peut voir sur les figures 8 et 9. De même pour les trois autres reines de ce joueur.

5.2 Boucle de jeu

Le déroulement de la partie est décrit par la fonction *game* dans le code fourni. Après l'initialisation des joueurs, la boucle principale de la partie commence. À chaque tour, un joueur est sélectionné pour jouer et le plateau de jeu est affiché. La partie se poursuit tant que la condition de fin de jeu n'est pas atteinte.

La première condition de fin de jeu est vérifiée lorsque toutes les reines d'un joueur sont bloquées, c'est-à-dire qu'il n'a plus de déplacements possibles. Pour chaque joueur actif, la fonction *right_move* est appelée pour vérifier la validité du mouvement effectué par le joueur. Si le mouvement est considéré comme incorrect, le joueur est marqué comme perdant, le compteur *player_in_game*

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

FIGURE 8 – Grille numéroté avec les indices des cases

	♔		♔	
♔				♔
♚				♚
	♚		♚	

FIGURE 9 – Grille initiale avec le placement des reines

est décrémenté et un message d'erreur est affiché. Cela garantit que les joueurs qui font des mouvements illégaux sont éliminés du jeu.

Si le mouvement est considéré comme correct, la fonction *update* est appelée pour mettre à jour l'état du plateau en fonction du mouvement effectué par le joueur. Ensuite, On vérifie si d'autres joueurs ont perdu à cause de ce dernier mouvement. La fonction *has_lost* est appelée pour vérifier si un joueur est bloqué, c'est-à-dire s'il ne peut plus effectuer de mouvements légaux. Si un joueur est bloqué, il est marqué comme perdant et le compteur *player_in_game* est mis à jour.

Après chaque tour, l'état du jeu, y compris les positions des reines et des flèches, est affiché en appelant la fonction *print_game*. Ce processus se répète jusqu'à ce que la condition de fin de jeu soit atteinte.

À la fin de la partie, le code détermine le gagnant en recherchant le dernier joueur qui n'est pas marqué comme perdant. Si tous les joueurs sont marqués comme perdants, cela signifie qu'il y a égalité et un message approprié est affiché. Sinon, le nom du gagnant est affiché.

Le code fourni implémente ainsi un mécanisme de jeu où les joueurs jouent tour à tour, vérifient la validité de leurs mouvements et sont éliminés s'ils effectuent des mouvements illégaux ou sont bloqués. Cela garantit un déroulement équitable du jeu et détermine le gagnant en fonction des règles du jeu spécifiées.

6 Stratégie de jeu choisie

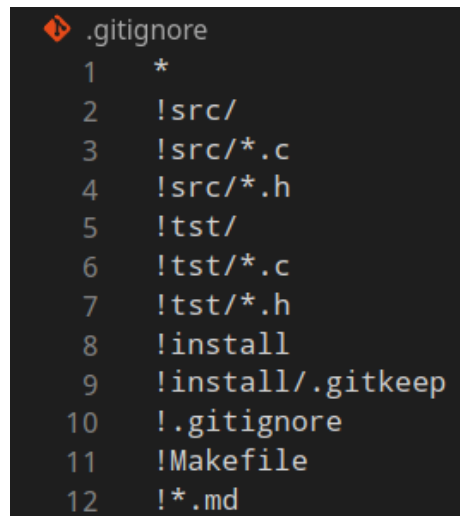
Le sujet imposait le développement de deux joueurs, nous avons développé l'idée d'un joueur qui ne fasse pas ses choix totalement au hasard pour ne pas renvoyer de move incorrect. Une phase de vérification de la possibilité du mouvement se fait alors dans le but de choisir un nouveau mouvement tant

que les précédents choix sont impossibles. Ce premier joueur est fonctionnel, en revanche nous n'avons pas réussi à corriger les erreurs de mémoire rencontrer avec le second joueur, même si la librairie est fonctionnelle. Pour celui-ci nous n'avons pas mis en place de calcul à l'aide d'heuristique, mais nous avons opté pour un choix moins aléatoire du positionnement des flèches. En effet, si une flèche se trouve en proximité d'une reine adverse, cette reine aura moins de possibilités de déplacement, ce qui n'apporte pas forcément d'avantage au début de la partie mais vers la fin peut être très avantageux pour notre joueur de réduire les possibilités de l'adversaire. Ainsi lors du choix de l'emplacement de la flèche, on analyse si le déplacement venant d'être effectué permet un tir aux alentours d'une reine adverse. Tout d'abord en observant les voisins immédiat de chaque reine adverse et en tentant de trouver un accessible.

7 Contrainte de projets et travail collaboratif

7.1 Git et Makefile

Étant quatre programmeurs, le développement git dit "workflow" en branches est très utile si ce n'est nécessaire. De plus vu le nombre de commit effectués un gitignore bien configuré était très intéressant. En effet, au lieu d'ignorer tous les fichiers spécifiques qui ne nous intéressent pas, on ignorent tous les fichiers et on garde uniquement les fichiers qui nous intéressent ! à l'aide des ! :



```
.gitignore
1  *
2  !src/
3  !src/*.c
4  !src/*.h
5  !tst/
6  !tst/*.c
7  !tst/*.h
8  !install
9  !install/.gitkeep
10 !.gitignore
11 !Makefile
12 !*.md
```

FIGURE 10 – Gitignore

On remarque que les fichiers de compilations ne sont pas inclus dans le dépôt, que les fichiers sources et les exécutables nécessaires le sont.

Le Makefile lui aussi a été conçu de façon intelligente. En effet il fournit des

règles pour compiler, exécuter et nettoyer le projet. Les principales fonctionnalités incluent : La définition des variables `GSL_PATH`, `LDFLAGS`, et `CFLAGS` qui spécifient respectivement le chemin d'installation de la bibliothèque `GSL`, les drapeaux de liaison pour l'édition des liens, et les drapeaux de compilation pour le compilateur. La création des cibles `all`, `build`, `test`, et `install` qui permettent de compiler les fichiers source, de générer les exécutables, de lancer les tests et d'installer les fichiers nécessaires. Les règles de compilation pour les fichiers source `.c` qui utilisent les variables `CFLAGS` et `LDFLAGS`. Les règles pour générer les bibliothèques partagées `libplayer.so` et `libplayer2.so` à partir des objets générés. La règle pour générer l'exécutable serveur en utilisant les objets générés et `server.o`. La règle pour générer l'exécutable `alltests` en utilisant les objets générés et `test_interface.o`. Les règles pour lancer le programme (`run`, `gdbrun`) et pour nettoyer le dossier (`clean`, `remake`, `valgrind`). Ces fonctionnalités permettent de compiler les fichiers source, générer les exécutables, exécuter le programme, nettoyer les fichiers temporaires et effectuer des opérations de débogage.

7.2 Répartition du travail

Afin d'éviter les conflits, on s'est divisé en deux groupes, chaque groupe travaillant sur un fichier. De même, au sein de chaque groupe on a testé le pair-programming. Il s'est avéré intéressant en effet par exemple on peut citer la notion abstraite et au premier abord non trivial de l'implémentation des matrices creuses au format `CSR`. Un élève a par exemple d'abord compris l'implémentation dans la bibliothèque `GSL` puis l'a expliqué à un camarade qui a lui même implémenté une fonction de recherche de voisins.

8 Conclusion

Ce projet a entraîné notre travail d'équipe, car il nous a incité à déléguer certaines tâches à son coéquipier. Il a aussi souligné l'importance de la communication, en effet on a été confronté à des difficultés car on ne savait pas quelles fonctionnalités avaient été implémentées exactement, ce qui a engendré des confusions dans la production de code. Mais au fur et à mesure des semaines, nous nous sommes améliorés pour aboutir à une production qui nous satisfait.